UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JOÃO PEDRO JENSEN OURIQUE

Análise Experimental do Desempenho de Bibliotecas de Ciência de Dados em Workflows com Alto Custo Computacional

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Lucas Mello Schnorr

Porto Alegre
2024

RESUMO

A limpeza, estruturação e subsequente análise de dados ganhou força enquanto disciplina nos últimos anos, sendo denominada de Ciência de Dados, o que se explica pelo valor que consegue trazer para empresas e instituições que a utilizam com eficácia. Com isso, cresceu também o volume de dados manipulado, gerando um custo computacional alto que deve ser administrado com inteligência pelos desenvolvedores da área. Neste trabalho, mesmo que a análise e identificação de padrões tenham papel significativo, estima-se que até 80% do tempo dos desenvolvedores é dedicado à exploração, limpeza e preparação de dados para uso em suas aplicações. Na medida que a complexidade dos workflows cresce, especialmente quando o volume de dados ultrapassa a capacidade de memória ou exige processamento intensivo, a escolha de ferramentas adequadas se torna essencial para garantir a eficiência e escalabilidade dos sistemas, tipicamente fazendo uso de ferramentas que utilizem soluções computação paralela. No entanto, há uma base de conhecimento limitada na literatura científica que explore comparações práticas e objetivas entre diferentes bibliotecas, notadamente em termos da diversidade dos recursos computacionais explorados, dificultando uma tomada de decisão informada. Este trabalho visa abordar essa limitação ao fornecer uma análise detalhada das ferramentas disponíveis para a Ciência de Dados, e assim trazer uma visão objetiva que suporte esta tomada de decisão em aplicações práticas.

Palavras-chave: Ciência de Dados. custo computacional. Preparação de Dados. Computação Paralela.

ABSTRACT

Data cleaning, structuring, and subsequent analysis have gained prominence as a discipline in recent years, now known as Data Science, due to the significant value it brings to companies and institutions that use it effectively. Consequently, the volume of manipulated data has also increased, leading to high computational costs that must be managed intelligently by developers in the field. In this line of work, although analysis and pattern identification play a significant role, it is estimated that up to 80% of developers' time is dedicated to exploring, cleaning, and preparing data for use in their applications. As the complexity of workflows grows, particularly when the data volume exceeds memory capacity or requires intensive processing, selecting appropriate tools becomes essential to ensure system efficiency and scalability, typically involving tools that utilize parallel computing solutions. However, there is a limited knowledge base in the scientific literature, especially in terms of the diversity of computational resources employed, that explores practical and objective comparisons between different libraries, making it difficult to make an informed decision. This work aims to address this gap by providing a detailed analysis of the available tools for Data Science, thereby offering an objective perspective to support decision-making in practical applications.

Keywords: Data Science. Computational Cost. Data Preparation. Parallel Computing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CPU	Central Processing Unit
CSV	Comma Separated Values
GPU	Graphical Processing Unit
JVM	Java Virtual Machine
PCAD	Parque Computacional de Alto Desempenho
RAM	Random Access Memory
RDD	Resilient Distributed Datasets
SSH	Secure Shell

SUMÁRIO

1 INTRODUÇÃO

A demanda por aplicações intensivas em dados cresceu significativamente nos últimos anos, ao passo que empresas dos mais diversos mercados percebem a necessidade de maior inteligência nos seus negócios e buscam tomar mais decisões baseadas em dados. Na criação de tais aplicações, Hellerstein, Heer and Kandel (2018) estimam que 80% do tempo dos desenvolvedores é empregado em Preparação de Dados: o processo de explorar, limpar e prepará-los para que estejam aptos a serem utilizados em aplicações.

Quando os workflows de tais aplicações aumentam em complexidade por lidar com um volume de dados que excede a capacidade de memória do sistema ou exigir operações que demandam mais tempo para serem processadas, torna-se necessário usar ferramentas que forneçam um melhor desempenho para criar sistemas eficientes e escaláveis. Neste contexto, os critérios utilizados para decidir por uma ferramenta variam: Nadi and Sakr (2023) concluem que cientistas de dados consideram como fatores mais importantes a usabilidade e o tamanho da comunidade e da documentação de cada biblioteca, além da segurança e do seu rigor estatístico. No entanto, não existe grande número de trabalhos de cunho científico que explorem esta questão trazendo observações práticas sobre como as variadas opções de biblioteca funcionam, dificultando uma tomada de decisão baseada em critérios objetivos além do que a documentação das próprias ferramentas informa e algum conhecimento difundido em fóruns das comunidades que as utilizam.

1.1 Objetivos

Desta forma, o objetivo deste trabalho é desenvolver e executar um método para uma análise comparativa do desempenho de ferramentas de criação de workflows de Ciências de Dados, utilizando a linguagem de programação python (ROSSUM; JR, 1995). Se discutirá o uso de paralelismo e seus benefícios em termos de escalabilidade, desempenho e eficácia no momento em que sua adoção passa a se justificar, para demonstrar a utilidade de tais ferramentas em soluções complexas de ciência de dados.

Sabendo que o trabalho de Preparação de Dados é bastante oneroso, almeja-se avaliar o desempenho de bibliotecas sob a ótica do tempo de execução, destacando as vantagens que cada ferramenta traz e em qual escala de grandeza podem reduzir tal tempo. Além do Pandas (MCKINNEY et al., 2011), que pode ser considerado a linha de base por ser a biblioteca mais utilizada (PETERSOHN et al., 2020a) em workflows de ciência

de dados, foram analisadas também as bibliotecas Polars (VINK et al., 2024), a API em python do Apache Spark (ZAHARIA et al., 2016), Vaex (BREDDELS; VELJANOSKI, 2018) e Modin (PETERSOHN et al., 2020b), este último utilizando Dask (ROCKLIN, 2015) como engine.

1.2 Contribuições

As contribuições deste trabalho foram viabilizadas com a utilização da estrutura do Parque Computacional de Alto Desempenho (PCAD) do Instituto de Informática da UFRGS (INFORMÁTICA, 2024). Execuções foram realizadas em sistemas computacionais com diferentes recursos, permitindo explorar o resultado trazido por essas bibliotecas em contextos distintos, ampliando a análise. O PCAD é uma infraestrutura computacional constituída de nós computacionais (servidores) interligados que podem ser utilizados conjuntamente para a execução de aplicações paralelas de maior porte, cujos recursos são acessados de maneira remota. Os resultados apresentados foram obtidos em três diferentes nós, cujas configurações serão detalhadas no Capítulo de Metodologia.

Espera-se então que esse trabalho possa servir de apoio na tomada de decisão de profissionais que se depararem com problemas semelhantes e possam tomar uma decisão objetiva dotados de resultados experimentais obtidos por meio da utilização das ferramentas em ambiente controlado.

1.3 Estrutura do Trabalho

O presente trabalho detalhará uma análise do desempenho de bibliotecas de ciência de dados em python, descrevendo a metodologia utilizada e os resultados encontrados. O Capítulo 2 apresenta a o aspecto teórico do estudo, destacando o DataFrame como estrutura de dados padrão em trabalhos de ciência de dados, as características de cada uma das bibliotecas utilizadas e seus principais diferenciais. Além disso, explora também outros trabalhos que tiveram objetos de estudo semelhantes e a perspectiva que foi adotada neste trabalho para que ele represente uma adição em relação a estes, por constituir uma análise isonômica de uma pipeline entre sistemas computacionais com características amplamente diferentes.

Em seguida, o Capítulo 3 apresenta a organização do trabalho prático desenvol-

vido, com as principais decisões de implementação tomadas, a caracterização do conjunto de dados, a maneira como as execuções foram coordenadas e os recursos computacionais utilizados. Por fim, discute a aplicabilidade dos resultados obtidos com tal metodologia, citando algumas limitações e uma análise crítica do trabalho desenvolvido.

Ainda, o Capítulo 4 traz a apresentação detalhada e discussão dos principais resultados obtidos por cada biblioteca, diferenciando-os segundo os sistemas computacionais utilizados. Por fim, o Capítulo 5 retoma sucintamente os principais aprendizados obtidos, citando também aspectos em que o trabalho poderia ser aprofundado posteriormente, caso sejam desenvolvidos novos trabalhos acerca do problema.

2 FUNDAMENTAÇÃO TEÓRICA

No presente capítulo se apresentará os principais conceitos relevantes para o experimento que segue, incluindo a caracterização das principais bibliotecas utilizadas e suas estruturas de dados.

2.1 Pandas e o DataFrame

As ferramentas analisadas neste trabalho utilizam como estrutura de dados fundamental o DataFrame, estrutura de dados tabular, potencialmente heterogênea e de tamanho mutável (TEAM, 2024) definida pela primeira vez pelos pesquisadores da Bell Laboratories Chambers, Hastie, e Pregibon como parte da linguagem de programação S (CHAMBERS; HASTIE; PREGIBON, 1990), que posteriormente daria origem à linguagem de programação estatística R (HENRY; WICKHAM, 2024). Por sua eficiência e flexibilidade, especialmente quanda comparado com bases de dados relacionais (WU, 2020), o DataFrame foi adotado e expandido em várias ferramentas e bibliotecas de ciência de dados, recebendo novos atributos e métodos concebidos para permitir a manipulação, filtragem e agregação de dados de maneira intuitiva.

O Pandas utiliza DataFrames desde sua primeira versão em 2009, embora sua implementação tenha mudado significativamente desde então, com a adição de diversos métodos, tendo como marco importante a versão 1.0.0 de 2020, que trouxe grande salto em robustez e consistência em sua API. Utilizando uma implementação em C/C++ no seu back-end para obter maior eficiência e capacidade de lidar com tipos de dados complexos, o Pandas utiliza o Apache Arrow (RICHARDSON et al., 2024), um formato colunar multiplataforma para armazenamento de dados em memória, como base para representar os seus tipos de dados desde sua versão 2, em conjunto com o Numpy (NADI; SAKR, 2023), uma vasta biblioteca para computação científica com suporte para operações em matrizes e vetores, esta utilizada desde as primeiras versões do Pandas.

Apesar da ampla utilização e de ser uma ferramenta bastante versátil, o Pandas apresenta uma limitação significativa em termos de escalabilidade devido à necessidade de manter os dados em memória. Ainda que esta abordagem ofereça vantagens significativas em termos de velocidade e facilidade de manipulação de dados, conjuntos de dados muito grandes que excedem a capacidade de memória disponível pode levar a problemas de desempenho, como atrasos significativos nas operações ou até mesmo a falhas de

execução devido à memória insuficiente. Para contornar essas limitações, as bibliotecas alternativas ao Pandas fazem uso de uma ou mais das seguintes técnicas:

- **Uso de múltiplas threads**: Como o Pandas é executado em thread única, naturalmente surgem esforços para utilizar paralelismo em múltiplas threads como uma maneira de acelerar a execução, tipicamente dividindo os DataFrames em partições que podem ser processadas em múltiplos núcleos de CPU.
- **Uso de Graphical Processing Units (GPUs)**: GPUs são otimizadas para realizar cálculo em paralelo por meio da vetorização de operações, possuindo milhares de núcleos menores do que os de uma CPU típica, motivando um grande número de usuários a utilizar esta tecnologia em aplicações de ciência de dados, como em algoritmos de Inteligência Artificial.
- **Otimização de Memória**: Bibliotecas fazem uso de técnicas de otimização como evitar a realização de cópias intermediárias em memória, como no caso do Vaex, e otimização do plano de execução como no exemplo do Spark, para obter ganhos de desempenho relevantes quando se trabalha com um grande conjunto de dados.
- **Avaliação tardia ou *lazy***: A técnica de Lazy Evaluation (Avaliação Tardia) é um modelo de execução onde a avaliação de operações é adicionada a um plano de execução e adiada até que seu valor seja estritamente necessário, contrastando com a avaliação gulosa (eager evaluation), onde as expressões são avaliadas assim que vinculadas a uma variável. Ela pode resultar em ganhos significativos de desempenho, por permitir a otimização do plano de execução, minimizar uso de memória e potencialmente evitar resultados desnecessários para o resultado final.
- **Sistemas em cluster**: Além de permitir paralelizar o trabalho entre os núcleos de uma máquina, bibliotecas também permitem a distribuição do processamento de dados entre um cluster de máquinas, aumentando a ordem de grandeza de poder computacional que pode ser usado pela aplicação.

Apesar do foco deste trabalho ter ficado sobre o python enquanto linguagem de estudo pela sua popularidade, versatilidade e mesmo pela familiariade do autor com a sua utilização, outras linguagens também tem utilização relevante no campo de ciência de dados, tanto no universo acadêmico quanto de mercado. Julia (BEZANSON et al., 2017) é uma linguagem de programação de alto nível e alto desempenho concebida para fazer uso de paralelismo em todos os níveis, seja via *multithreading*, GPUs ou por computação distribuída. Apesar de recente, vem ganhando bastante tração, especialmente em áreas de

pesquisa e desenvolvimento que exigem alto desempenho computacional. O R, tradicional linguagem de computação estatistíca, possui seu próprio ecossistema de bibliotecas para lidar com DataFrames (WICKHAM et al., 2023) e é citada como uma linguagem mais madura e mais bem curada do que o python (PETERSOHN et al., 2020a), além de se manter entre as 10 linguagens de programação mais populares do mundo (PYPL, 2024).

2.2 Além do Pandas: bibliotecas com maior escalabilidade

Considerando as limitações identificadas no Pandas e as funcionalidades que possibilitam superá-las, cada biblioteca que se apresenta como uma opção mais escalável que o Pandas faz uso de alguma otimização ou técnica mais sofisticada para alcançá-lo. na A Tabela 2.1 reúne um resumo das principais funcionalidades que as bibliotecas utilizadas neste trabalho implementam como formas de obter melhorias de performance. Outras técnicas como o uso de sistemas em Cluster e de GPUs não foram exploradas, e ambas destacam-se como horizontes em que este estudo poderia ser aprofundado, como discutido na Seção 5.2.

Tabela 2.1: Funcionalidades oferecidas por cada biblioteca utilizada

	Pandas	PySpark SQL	Modin	Polars	Vaex
Uso de múltiplas threads		x	x	x	x
Otimização de Memória		x	x	x	x
Avaliação lazy		x		x	x
Linguagem Nativa	Python	Scala	Python	Rust	C e Python
Outros requisitos		SparkContext	Dask		
Versão utilizada	2.1.4 (12/2023)	3.4.1 (06/2023)	0.26.1 (01/2024)	0.19.19 (12/2023)	4.17.0 (07/2023)

Como visto em:(MOZZILLO et al., 2023)

2.2.1 Apache Spark

O Apache Spark é um sistema de computação em cluster que oferece uma interface para processamento de dados distribuído, escrito originalmente em Scala (ODERSKY et al., 2004), com uma API que permite sua utilização em python, a PySpark. Sua imple-

mentação se baseia em uma abstração chamada de SparkContext, que roda sobre o nó principal da execução do programa e age como um gerenciador de todas as funcionalidades do cluster (SHAIKH et al., 2019), gerenciando o uso de threads e instanciando uma máquina virtual Java (JVM), que executa em paralelo, se comunicando com os processos em python ao longo da execução. Os dados são representados internamente se utilizando do conceito de *Resilient Distributed Datasets*(RDDs), uma coleção de elementos sobre os quais podem ser realizadas operações de maneira distribuída com tolerância a falhas, tanto a partir de uma coleção definida no programa original quanto por meio de uma referência a um conjunto de dados localizado em um sistema de armazenamento externo (ZAHARIA et al., 2012). Além disso, as operações sobre os dados são orquestradas por meio de um otimizador, chamado de Catalyst, que faz uso de contruções da linguagem Scala para gerar planos de execução otimizados. Embora seja uma ferramenta popular em sistemas em cluster, seu uso em uma única máquina também é justificado, por permitir a distribuição de tarefas entre diferentes núcleos de processamento. O Apache Spark possui um módulo que permite integrar processamento relacional à sua programação procedural através da sua API de DataFrames, o SparkSQL (ARMBRUST et al., 2015), que foi a API de operadores escolhida para ser utilizada neste trabalho.

2.2.2 Modin e Dask

O Modin é uma biblioteca projetada para permitir a adoção de processamento paralelo com mínima necessidade de adaptação (PETERSOHN et al., 2020a), citando uma compatibilidade acima de 90% com a API de DataFrames do Pandas (DEVELOPERS, 2024). Por ter arquitetura modular, permite a utilização de engines no seu backend, como Dask (ROCKLIN, 2015) e Ray (MORITZ et al., 2018). Neste trabalho foi utilizado a sua versão com Dask, uma biblioteca de computação paralela projetada para integrar-se de forma fluida com as bibliotecas existentes do ecossistema de ciência de dados em Python como NumPy, Pandas e scikit-learn, que surgiu do reconhecimento do gargalo que estas ferramentas introduzem ao serem limitadas pela memória RAM. Foi desenvolvida para preencher a lacuna entre o processamento em lote de grandes conjuntos de dados, típico de sistemas como Apache Hadoop e Apache Spark, e a computação interativa em memória, permitindo análises de dados em larga escala em Python sem a necessidade de recorrer a sistemas de processamento de dados distribuídos mais complexos e menos flexíveis. Para isso, o Dask oferece estruturas de dados paralelas que estendem interfaces

comuns como arrays do NumPy, DataFrames do Pandas e listas do Python.

2.2.3 Vaex

O Vaex foi projetado para ser capaz de realizar análises sobre grandes conjuntos de dados, utilizando algoritmos de memória externa, concebidos para lidar com conjuntos de dados que excedem a capacidade da memória RAM, e trazendo otimizações como avaliação *lazy* de operações, arquivos mapeados em memória e uma política de não-criação de cópias dos dados em memória (BREDDELS; VELJANOSKI, 2018). Em adição ao seu módulo *core* para as operações de computação e processamento dos dados, o Vaex possui uma gama de módulos para diferentes necessidades comuns a aplicações de ciência de dados, como visualização, comunicação cliente-servidor e até mesmo operações concebidas para lidar com conjuntos de dados de astronomia. Além disso, mantém uma API bastante similar ao Pandas, reduzindo o esforço necessário para migrar para a ferramenta.

2.2.4 Polars

Polars é uma biblioteca *multi-threaded* implementada em Rust utilizando Apache Arrow como seu modelo de memória, dotada de um otimizador de queries capaz de reduzir uso desnecessário de memória. Seus desenvolvedores criaram uma Linguagem de Domínio própria, baseda em dois componentes: *Contexts*, o contexto em que uma expressão precisa ser avaliada, e *Expressions*, uma árvore de operações que descreve como construir uma série de dados (POLA.RS, 2024). A biblioteca suporta avaliação *lazy* e *eager* das operações, apesar de recomendar a primeira quando alto desempenho é desejado, pelo potencial de otimização que ela traz. Ademais, no momento em que este documento é escrito, operações em streaming são parcialmente suportadas por alguns operadores, permitindo a execução em conjuntos de dados que excedem a capacidade da memória por processá-los em lotes, sendo que a perspectiva é de que isto seja adicionado a novos operadores futuramente.

2.3 Trabalhos Relacionados

O estudo de soluções que tragam alto desempenho a aplicações de ciência de dados é parte relevante da pesquisa realizada nos últimos anos. Modelos de implementação mais robustos são concebidos para ganhar em escala e permitir a exploração de conjuntos de dados maiores (SINTHONG; CAREY, 2019) (PETERSOHN et al., 2020a). A necessidade por esta maior capacidade computacional inspira comparações de bibliotecas na sua utilização em conjuntos de dados enormes classificados como *big data* (STANčIN; JOVIć, 2019).

Além disso, a etapa de Preparação de Dados, por sua relevância a *pipelines* de todo genêro, é por si só objeto de estudo, no que tange aos padrões e procedimentos adotados em diferentes produtos (FERNANDES et al., 2023), assim como das principais ferramentas utilizadas neste processo, os principais operadores envolvidos e sua aplicabilidade (HAMEED; NAUMANN, 2020a). No entanto, pouco trabalho científico é voltado para explorar o momento em que se torna necessário dar os primeiros passos em direção a soluções mais escaláveis, quando a solução mais tradicional começa a apresentar problemas e é preciso escolher entre as variadas soluções, tendo em vista seus próprios objetivos, limitações e recursos, sendo as primeiras opções de informações muitas vezes limitadas a blogs e fóruns com pouco rigor científico (SCHMITT, 2024).

Mozzillo et al. (2023) exploram esta questão como objetivo do seu trabalho, apresentando-o como a primeira comparação experimental rigorosa de bibliotecas de DataFrames em tarefas de Preparação de Dados. O presente trabalho adota abordagem semelhante, visando destacar quais vantagens pode-se esperar da adoção de uma das bibliotecas apresentadas, destacando o momento em que tipicamente esta decisão se faz necessária. Além disto, a utilização da plataforma computacional com recursos diversos do PCAD permite, além de poder ser utilizada para fins de reprodutibilidade do estudo de Mozzillo e de suas conclusões, trazer uma análise mais abrangente, possibilitando a obtenção de resultados em sistemas com uma vasta gama de recursos computacionais.

3 METODOLOGIA

Neste capítulo, apresenta-se o método utilizado para gerenciar as execuções, destacando-se a pipeline desenvolvida e como se deu sua implementação, os sistemas computacionais utilizados e o conjunto de dados escolhido como objeto de estudo. Discutem-se também algumas limitações do trabalho desenvolvido, bem como justificativas para as decisões tomadas ao longo do desenvolvimento.

3.1 Visão geral

Visto que o objetivo do trabalho é comparar o desempenho de diferentes bibliotecas, foi necessário uma interface de comparação que possibilitasse executá-las de maneira compatível e parametrizável, viabilizando a extração e análise de resultados obtidos de maneira padronizada. O Bento (MOZZILLO; GAGLIARDELLI, 2024), framework de código aberto desenvolvido por Mozzillo et al. (2023) foi criado para viabilizar comparações como esta, possibilitando definir e implantar worklows de ciência de dados em qualquer biblioteca da linguagem python ao criar uma interface comum entre suas APIs, executando-as em ambientes controlados através de contâineres Docker. Criando-se um fork do projeto (OURIQUE, 2024) e realizando algumas adaptações, foi possível adequá-lo às necessidades deste trabalho e implementar uma pipeline de execução sobre o conjunto de dados *Airline on Time* (EXPO, 2009).

Tomando como base os principais preparadores definidos por Hameed and Naumann (2020b), foram implementadas 24 operações, como consta na Tabela 3.1, acompanhados do método que os implementa no Pandas para referência, todos da API do DataFrame, exceto onde indicado. Estes foram escolhidos para representarem uma pipeline hipotética representativa de um caso prático, e são classificados segundo a etapa do processo em que são aplicados, seguindo a mesma definição de etapas usada por Mozzillo et al. (2023): Entrada/Saída (I/O), Análise Exploratória (EDA), Transformação de Dados (DT) e Limpeza de Dados (DC).

Utilizando o Bento, é possível realizar experimentos em três modos de execução, sendo que em cada um deles se extraiu o tempo de execução e memória utilizada: da Pipeline inteira, por Etapa ou por Preparador. Esta diferenciação busca trazer resultados mais completos e permite avaliar o impacto de otimizações como a avaliação *lazy*, uma vez que a extração de métricas por Preparador por exemplo, força a realização dos cál-

Tabela 3.1: Preparadores utilizados por etapa

Etapa	Preparador	Pandas API
I/O	load_dataset	read_csv
	output_dataset	to_csv
EDA	get_columns	columns (propriedade)
	get_stats	describe
	locate_null_values	isna
	locate_outliers	percentile (NumPy)
	search_by_pattern	str.contains + re.compile
	sort	sort_values
	query	query
DT	cast_columns_types	astype
	delete_columns	drop
	rename_columns	rename
	calc_column	where (NumPy)
	join	merge
	pivot	pivot_table
	one_hot_encoding	get_dummies + concat
	categorical_encoding	astype
	groupby	groupby
DC	fill_nan	fillna
	delete_empty_rows	dropna
	replace	replace
	change_num_format	round
	set_content_case	str.lower/str.upper/str.title
	edit	apply

culos intermediários após cada operação. No entanto, considerou-se neste trabalho que a medida mais representativa do desempenho de cada biblioteca é a obtida ao executar a pipeline inteira, por ser o resultado que mais interessaria em uma aplicação prática.

Para extração da métrica de tempo de execução de cada experimento utilizou-se do módulo *time* da distribuição padrão do python. Há de se fazer a ressalva de que, em algumas máquinas, nem todas as bibliotecas executaram sem falhas ao processar todo o conjunto de dados, casos em que destacou-se na descrição dos resultados o número de registros máximo em que elas foram capazes de ter sucesso. Além disso, extraíram-se também medidas de picos de memória por meio da biblioteca *tracemalloc*, que também faz parte da distribuição padrão do python e é reconhecida como uma das principais opções para realização de tais medições (BLANCO et al., 2024). Considerou-se esta uma medida adequada por representar o mínimo de memória necessário para realizar cada execução.

Para permitir a parametrização de cada Preparador e contemplar as diferenças nas APIs de cada biblioteca, utiliza-se um arquivo *json* para declarar os métodos da pipeline, os argumentos utilizados por cada um e eventuais comandos extras que podem ser necessários. Para o preparador *join*, por exemplo, estes comandos adicionais permitem a leitura de outro DataFrame a partir de um arquivo csv secundário. A Figura 3.1 exemplifica como foi parametrizado o método *calc_column*. Através de uma interface comum que permite a determinação de uma nova coluna a partir da combinação de colunas existentes através de uma expressão, define-se quais parâmetros serão passados para uma execução padrão através da chave *input*, e também os parâmetros customizados para a biblioteca Polars, acomodando diferenças sutis de sintaxe de uma biblioteca em relação às demais.

Figura 3.1: Parametrização do preparador calc_column

```
{
      "method": "calc_column",
      "input": {
        "col_name": "NewCancelled",
        "columns": ["Cancelled", "CancellationCode"],
        "f": `lambda x: 'No' if x[1] == '' or x[0] == '' else '
           Yes'`
      },
      "input_polars": {
        "col_name": "NewCancelled",
        "columns": ["Cancelled", "CancellationCode"],
        "f": `lambda x: 'No' if x['CancellationCode'] == '' or x
           ['Cancelled'] == '' else 'Yes'`
      },
}
```

3.2 O Conjunto de Dados

O trabalho utilizará como estudo de caso uma base de dados de aproximadamente 120 milhões de registros de voos domésticos nos Estados Unidos entre 1987 e 2008 (EXPO, 2009), além de arquivos secundários com dados complementares como informações sobre as aeronaves utilizadas, aeroportos e companhias aéreas. Tal conjunto foi escolhido por seu volume significativo, permitindo reproduzir um custo computacional relevante, e se encontra em arquivos *csv* comprimidos, ocupando 1,6 GB de memória, que se transforma em 12 GB quando descomprimidos. A Tabela 3.2 reúne as principais características do conjunto de dados analisado. Ao longo dos experimentos no PCAD,

execuções foram realizadas com um subconjunto dos dados, começando por 10^6 registros, passando para 10^7 registros e aplicando incrementos de 10^7 até que se considerasse o conjunto inteiro.

Nas execuções com o PC, uma vez que a maioria das bibliotecas falhou com subconjuntos menores dos dados, complementou-se esta abordagem ao também começar com 10^6 registros, mas utilizando incrementos de 2,5 * 10^6 até chegar a 2 * 10^7 registros, quando o incremento foi alterado para os mesmos 10^7 registros utilizados na outra análise. Desta forma, foi possível analisar com detalhes a área de transição em que algumas bibliotecas se tornaram inviáveis quando menos recursos computacionais estavam disponíveis.

Tabela 3.2: Resumo das características do Conjunto de Dados

	Airline on time Data
Tamanho CSVs (GB)	12
Tamanho CSVs comprimidos(GB)	1,6
Nº Colunas	29
Nº Linhas (registros)	1,2 * 10^8
Colunas Num - String - Bool	19 - 8 - 2
% Nulos	21%
Comprimento máximo de string	9

3.3 Sistemas Computacionais utilizados

Para servir como linha de base para os experimentos, um computador pessoal (PC) com 20 GB de memória RAM foi utilizado, representando o cenário mais corriqueiro em que um usuário que trabalhe com workflows cujo custo computacional comece a tornar-se um problema e portanto faça-se necessário maior desempenho. Além disso, acessando remotamente a estrutura do PCAD via SSH, foram utilizados os servidores Tupi, Cidia e Blaise que, com capacidade desde 128GB até 320 GB de RAM, possibilitaram reproduzir análises em máquinas com alta capacidade de desempenho, como apresentado na Tabela 3.2.

Tabela 3.3: Especificações de cada ambiente utilizado

	PC	Tupi	Blaise	Cidia
Núcleos	2	24	44	16
Threads	4	32	88	32
Memória RAM	20 GB DDR4 2133 MT/s	128 GB DDR4 4000 MT/s	256 GB DDR4 2400 MT/s	320 GB DDR4 2400 MT/s
Processador	Intel® Core™ i5-7200U CPU @ 2.50GHz × 4	Intel i9-14900KF (Q4'23), 3.2 GHz	2 x Intel Xeon E5-2699 v4 Broadwell (Q1'16) 2.2 GHz	2 x Intel Xeon Silver 4208 Cascade Lake (Q2'19) 2.1 GHz

3.4 Limitações e Mitigação de Riscos

Para garantir execuções homogêneas e poder portar o código desenvolvido entre os diferentes sistemas, o Bento é implementado e executado a partir de imagens Docker, tanto no PC quanto nos servidores do PCAD. A cada experimento lançado, diversas execuções foram feitas de maneira sucessiva parametrizando-se os modos de execução (medição da pipeline completa ou de cada etapa e método), bibliotecas e tamanho do problema desejados, seguindo a seguinte estrutura: para um dado modo de execução, inicia-se por um tamanho de problema, executando todas as bibliotecas, e em seguida passando ao próximo tamanho, seguindo desta forma até finalizar todas as execuções de um dado modo, seguindo ao próximo, caso tenha sido definido para o experimento em questão.

No caso do PCAD, o acesso se dá por meio de ssh e as execuções são realizadas através de um gerenciador de filas Slurm (YOO; JETTE; GRONDONA, 2003) que, combinado com scripts bash, permite o uso de cada um dos nós computacionais por um tempo pré-definido de maneira dedicada e sem possibilidade de perturbações por outros processos, associando execuções a um ID único e gerando arquivos de logs para depuração e rastreabilidade de cada experimento.

Este trabalho apresenta uma visão abrangente da utilização de 5 das principais opções de bibliotecas disponíveis para criação de workflows de ciência de dados com a linguagem python. No entanto, reconhece-se que algumas abordagens não foram exploradas, como a utilização de GPUs e a execução em cluster, citadas no Capítulo 5 como próximos passos que poderiam ser tomados para enriquecer esta análise. Além disso, a utilização de diferentes máquinas permite a observação do uso do paralelismo em diferentes contextos, destacando sobretudo a influência da memória de processamento nesta

discussão. No entanto, uma limitação que pode ser apontada é a grande diferença entre a memória de processamento da máquina PC e as demais, separadas por um fator de pelo menos 6: 20GB e 128GB na Tupi, a máquina do PCAD com menor memória dispoível. Ampliar a análise com uma máquina cujos recursos estivessem em um intermédio entre estas duas poderia trazer mais contexto sobre a influência que o aumento da memória pode ter sobre a escolha entre as bibliotecas.

Ademais, o exercício de fazer uma análise objetiva do desempenho de cada uma das bibliotecas de maneira equivalente é complexo, uma vez que cada uma possui suas próprias técnicas de otimização que podem visar aprimorar determinado operador mais do que outro. Apesar de utilizar técnicas de virtualização para manter portabilidade e um framework comum para executar cada biblioteca de maneira isonômica, a implantação de cada operador ainda é específica, variando ligeiramente em função da API de cada biblioteca, estando sujeita às decisões do autor, segundo sua familiaridade com a ferramenta e a documentação disponível. Como apontado na Sub-seção 4.1.1, existe margem para que existam grandes variações nos resultados em função de tais decisões de implementação, mesmo que não necessariamente isso se deva a uma má interpretação da documentação da ferramenta. Não obstante, a presente análise representa o melhor esforço do autor em fazer bom uso de cada ferramenta, se apoiando na documentação oficial disponível, no seu próprio conhecimento e em outras fontes de informação disponíveis na internet.

Por fim, a presente análise utiliza como caso de estudo um conjunto de dados específico, e portanto suas características devem ser levadas em consideração na observação dos resultados. Aspectos como o formato utilizado como entrada, a proporção entre número de linhas e colunas, o número de elementos nulos, strings e valores númericos podem influenciar os resultados finais; portanto o uso das conclusões apresentadas em outros contextos deve ser feito considerando a semelhança a este estudo. Isto posto, este trabalho pode certamente ser bem utilizado como uma referência geral na escolha de bibliotecas de ciência de dados em workflows com alto custo computacional.

4 RESULTADOS

Na análise realizada, o principal objetivo é propor uma abordagem para avaliar as vantagens que cada biblioteca traz na execução de workflows com alto custo computacional, reproduzindo a decisão que um desenvolvedor precisaria tomar ao identificar que o Pandas não está atendendo suas necessidades. Desta forma, o principal interesse é avaliar a execução da pipeline completa, avaliando o tempo de execução mas também a eficiência no uso de recursos computacionais e sob quais condições cada biblioteca falha, deixando de produzir execuções bem sucedidas.

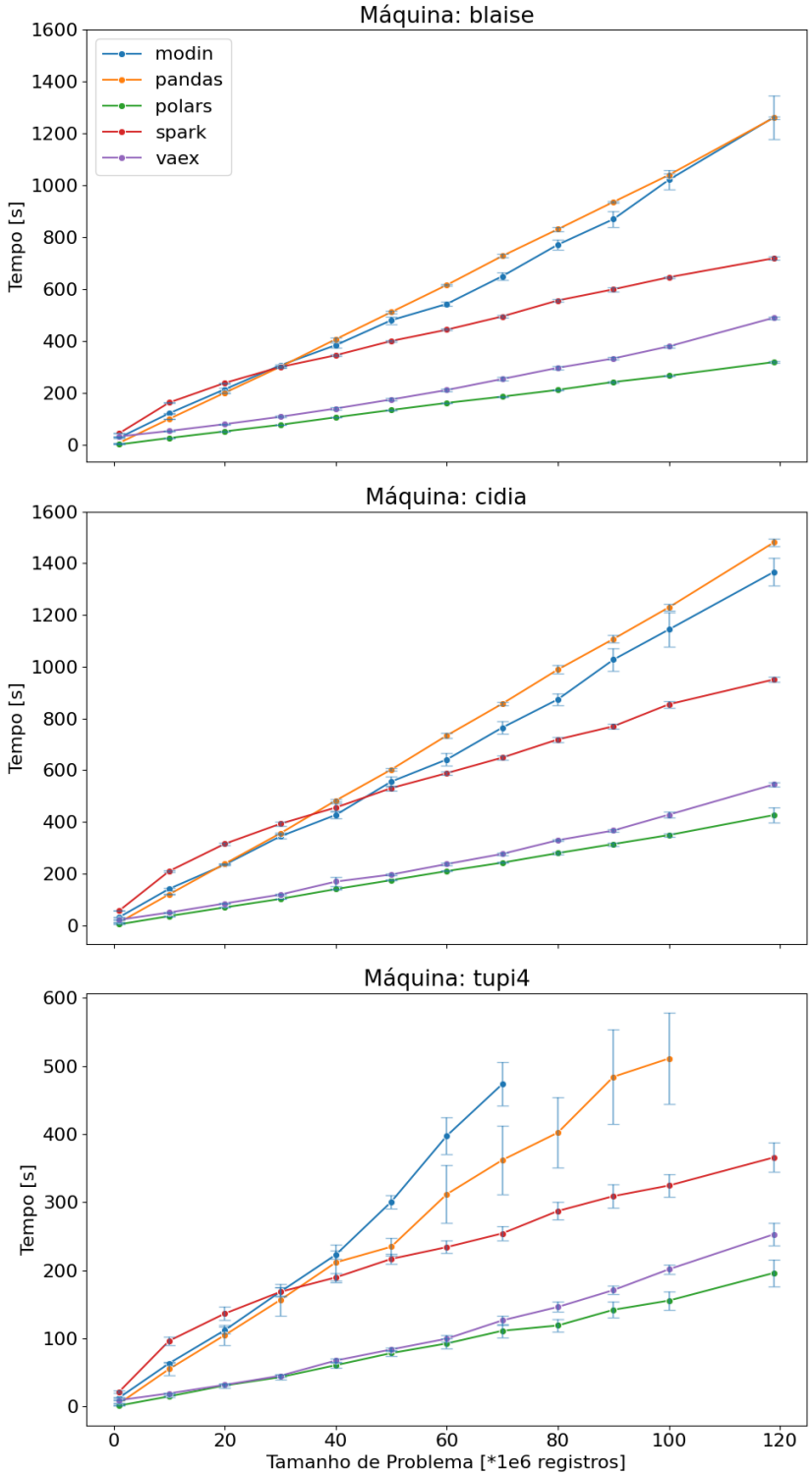
4.1 Resultados da execução completa

A Figura 4.1 reúne os resultados obtidos para a execução da pipeline completa, em cada máquina utilizada. Por possuir o processador com a maior frequência de clock, a Tupi foi a máquina que concluiu as execuções mais rapidamente, seguida pela Blaise e, por último, a Cidia. Por outro lado, por possuir a menor memória, foi a única em que nem todas as bibliotecas foram capazes de processar o conjunto de dados inteiro, com o Modin e o Pandas falhando. Próximos a estes pontos de falha na Tupi, observou-se também a maior variabilidade nas medidas de tempo obtidas para as duas bibliotecas. Quanto aos melhores desempenhos, percebe-se que o Polars e o Vaex destacaram-se, produzindo consistentemente os melhores resultados em termos de tempo de execução. Assim como estas duas, o Spark também obteve bons resultados, mas levando 86% mais tempo do que o Polars e 46% mais tempo do que o Vaex na Tupi para o conjunto de dados inteiro, o que também foi observado, embora em graus diferentes, na Cidia (132% e 78% mais tempo do que o Polars e o Vaex, respectivamente) e na Blaise (120% e 45%).

Por outro lado, o Modin apresentou resultados insatisfatórios, obtendo tempos de execução muito próximos ao Pandas e até falhando antes dele na Tupi, o que é inesperado para uma biblioteca que visa oferecer aumento de desempenho em relação ao Pandas por implementar técnicas de otimização e paralelismo. Tendo em vista estes resultados inesperados, algumas tentativas foram feitas para depurar os problemas desta biblitoeca, e estas são relatadas com mais detalhes na Sub-seção 4.1.1.

Enquanto isso, o Pandas apresentou resultados que demonstraram sua competitividade frente às alternativas que fazem uso de paralelismo quando há memória suficiente para a execução, ainda que seja possível identificar os ganhos de desempenho trazidos por

Figura 4.1: Resultados das execuções no PCAD por tamanho do conjunto de dados, por máquina



suas alternativas. Na Tupi, foi possível executar a pipeline com até 10^8 registros, ponto a partir do qual as execuções resultaram em erros de estouro de memória. Usando a média das execuções neste tamanho de problema como referência, o Pandas teve tempo 57%

maior do que o Spark, 153% maior do que o Vaex e 228% maior do que o Polars. Esta relação de ordem se manteve nas máquinas Cidia e Blaise, sem grande variação na ordem de grandeza das diferenças entre cada biblioteca.

4.1.1 Dificuldades com o Modin

Como já ressaltado na Sub-seção 2.2.2, o Modin é uma biblioteca concebida para ser uma alternativa ao Pandas, trazendo melhor desempenho e apresentando o importante atrativo de ser uma alternativa de fácil adoção, mantendo alta compatibilidade com a API do Pandas, ao ponto de citar que apenas a troca do *import* da API do Dataframe do Pandas pelo Modin seria suficiente para trazer melhores resultados (MODIN, 2024a).

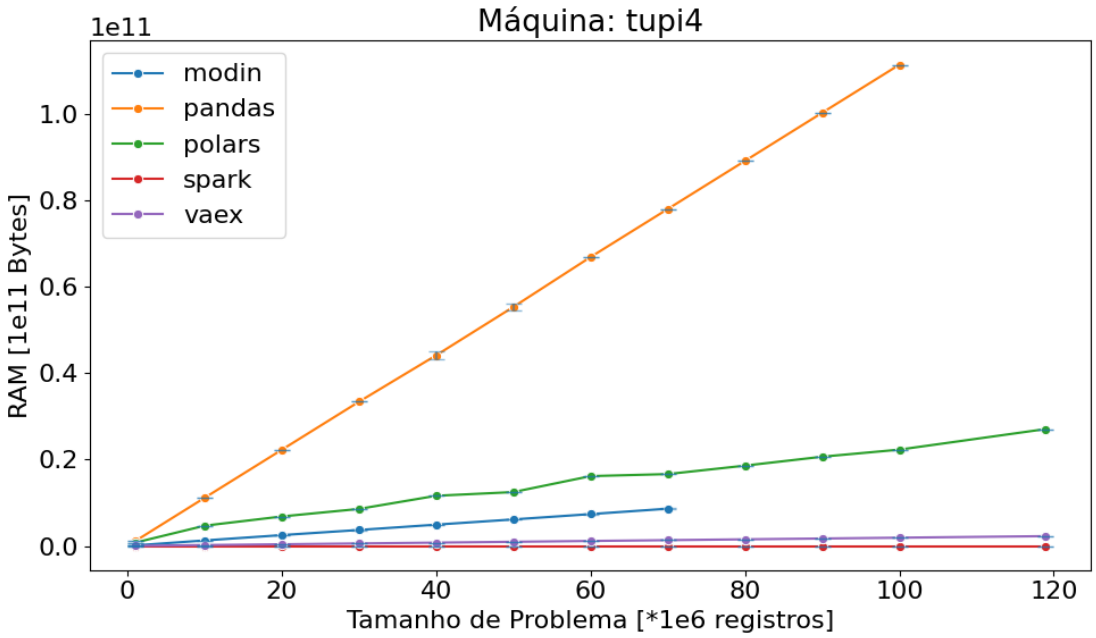
No entanto, à medida que resultados começaram a ser obtidos para este trabalho, as medições constatadas para o Modin se mostraram piores do que seria esperado para a biblioteca: inicialmente a mesma trazia pior desempenho do que o Pandas. Assim, diversas tentativas foram feitas de localizar possíveis problemas e melhorar o desempenho: o monitoramento do uso de recursos de CPU ao longo das execuções confirmou o uso de múltiplos processos, atestando a existência de paralelismo. Uma vez que a instalação e a configuração da biblioteca e do Dask são bastante simples ao consultar a documentação, descartaram-se erros neste âmbito. Da mesma forma, diferentes implementações dos preparadores também foram testadas, mas por ter de fato API muito semelhante ao Pandas e não haver grande ambiguidade sobre seu uso, não considerou-se que pudessem ser fonte de problemas. Uma exceção foi a alteração da função de leitura: uma vez que, para cada execução, lia-se apenas um subconjunto dos dados, o parâmetro *nrows* da função read_csv (MODIN, 2024b) estava sendo utilizado, até que foi constatado que seu uso impactava significativemente o tempo de execução: ao removê-lo e passando a descartar linhas após a leitura, removendo-as do DataFrame, este tempo se reduziu por um fator próximo de 3, resultado surpreendente por não ocorrer em outras bibliotecas e nem haver qualquer menção a este comportamento na documentação da biblioteca. Este resultado trouxe o desempenho do Modin para praticamente o mesmo patamar do Pandas, e as tentativas posteriores de melhorá-lo não foram bem sucedidas, o que demonstra a dificuldade de utilização de tais bibliotecas, tanto no uso prático quanto na criação de uma comparação objetiva de seu desempenho frente às outras.

Ademais, foram feitas algumas tentativas de alterar os valores padrão de parâmetros de inicialização do Cluster Dask local como: o número de workers, o número de

threads por worker, o uso de threads ou processos como a técnica de paralelismo, além de alterar o número de partições em que o DataFrame é dividido, todas sem sucesso. Ademais, por possuir arquitetura modular, o Modin permite a utilização de diferentes *engines* na sua execução; portanto, foi feita a tentativa de alterar a engine utilizada de Dask para Ray, como uma forma de verificar se não haveria algum problema na configuração ou uso da engine. Por não resultar em melhoria de desempenho, esta hipótese foi descartada.

Além dos tempos de execução, o fato da biblioteca falhar antes do Pandas também surpreendeu, considerando que sua proposta é justamente trazer maior escalabilidade ao Pandas. Em termos da causa de falha, os logs das execuções não apontaram estouro de memória, finalizando apenas com o status de erro genérico 1. De fato, as medições de memória obtidas por meio da ferramenta tracemalloc e apresentadas na Figura 4.2, atestam que o uso de memória ficou bem abaixo do Pandas, como seria de se esperar.

Figura 4.2: Uso de memória RAM na máquina Tupi

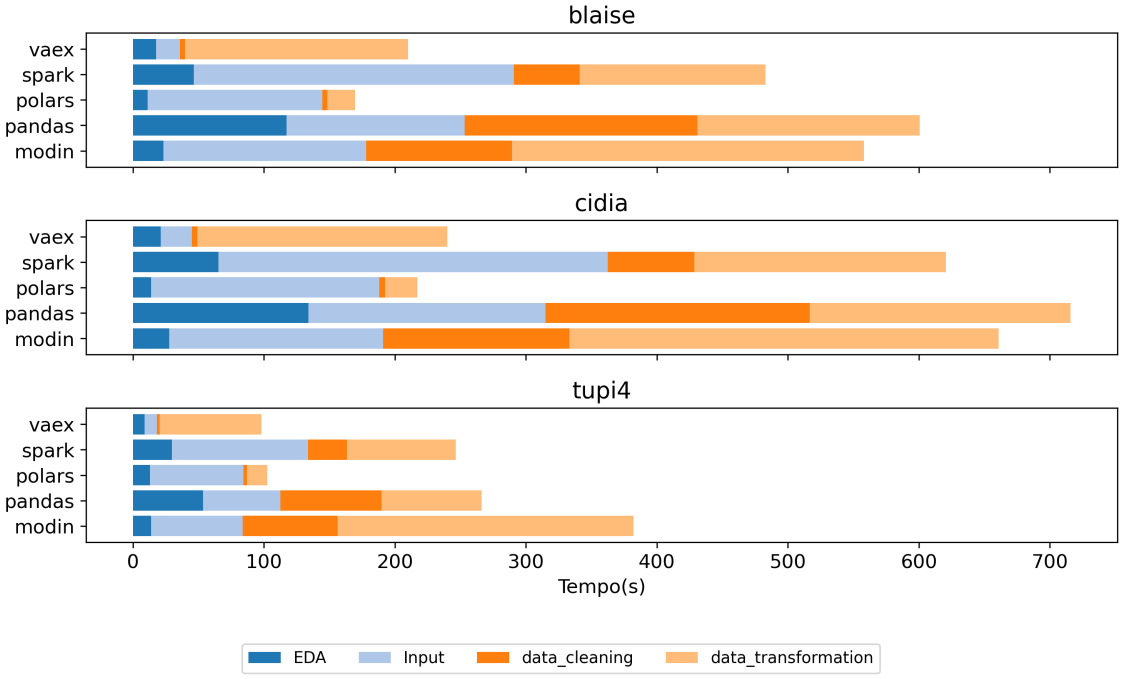


Havendo tempo limitado para explorar mais a questão, decidiu-se por encerrar a tentativa de melhorar os resultados obtidos com a biblioteca. A origem do problema não pôde ser determinada com precisão, além de aparentar ter relação com a comunicação entre os workers, apesar de que as mensagens de erro não davam muito mais contexto do que isso. O relato aqui apresentado visa dar mais detalhes sobre resultados imprevistos, buscando possibilitar a reprodutibilidade do problema em outros contextos, além de sensibilizar outros usuários de que a utilização e também a avaliação destas bibliotecas não é trivial.

4.2 Análise das Diferentes Etapas

O foco da análise apresentada é de obter, por meio de execuções de uma pipeline completa, uma referência para o desempenho de cada biblioteca, possibilitando uma análise comparativa. Naturalmente, alguns preparadores possuem papel mais significativo na análise por levarem mais tempo, e tais resultados também variam entre as bibliotecas. A Figura 4.3 traz os resultados por etapa da pipeline, segundo classificação das etapas detalhada na Tabela 3.1, usando como referência as execuções aplicadas sobre 6 * 10^7 registros, referência escolhida por ser o maior tamanho de problema em que não houve falhas para o Modin e Pandas na Tupi.

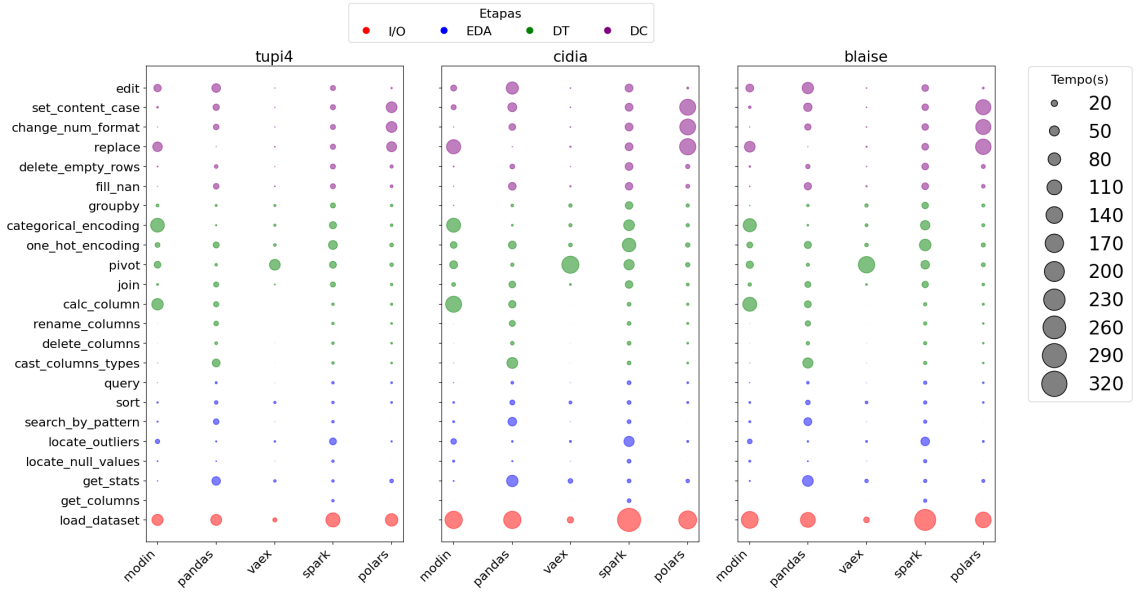
Figura 4.3: Tempo de execução por etapa, considerando 6 * 10^7 registros



Percebe-se que os gargalos na execução variam entre as diferentes bibliotecas. Para o Spark e o Polars, percebe-se pela barra azul mais clara que a etapa de leitura tem contribuição bastante significativa, representando cerca de 40% e 80% do tempo de execução para cada um, respectivamente . Esta medida se reduz no Modin, no qual a etapa de leitura torna-se a segunda mais significativa após a DT, e o mesmo ocorre no Vaex, que se mostrou extremamente rápido em relação às outras bibliotecas em todas as etapas a não ser pela de DT, que ficou entre 60% e 80% do tempo total de execução nas diferentes máquinas. Enquanto isso, para o Pandas o tempo é mais uniformemente distribuído entre as etapas.

Trazendo esta análise em nível mais detalhado, a Figura 4.4 mostra o tempo de cada preparador discriminado por máquina e biblioteca, onde é possível identificar qual deles foi mais significativo em cada caso. Nota-se novamente a relevância do tempo de leitura dos dados, sendo aquele que mais demorou para o Polars, o Spark e o Pandas em todas as máquinas. Dados estes resultados, uma análise à parte foi destinada à leitura de dados na Seção 4.3.

Figura 4.4: Tempo de execução por preparador, considerando 6 * 10^7 registros



Além disso, operações de modificação de muitos valores ou da criação de novas colunas a partir de uma nova coluna foram os principais garagalos de execução no Polars e Modin, com o replace, edit e calc_column. Já com o Pandas e o Spark, a distribuição de tempo de execução foi mais homogênea entre os preparadores, à exceção do método de leitura dos dados, que foi o mais demorado. Um resultado particular ocorre no Vaex, onde a grande maioria dos preparadores foi executada muito rapidamente, a não ser pelo outlier pivot, que tomou mais da metade do tempo de execução, enquanto para as outras bibliotecas isto não se observou para este preparador. Isso é explicado pelo fato de que, por não estar disponível na API do DataFrame Vaex diretamente, sua implementação foi realizada utilizando outros preparadores, o que certamente impactou no desempenho da biblioteca nesta operação. Portanto, definido-se uma pipeline onde este preparador não estivesse incluído, os resultados para o Vaex seriam bastante superiores.

4.3 Lidando com I/O nas pipelines

O problema de ingerir e extrair dados pode ser resolvido de diversas maneiras, seja pela conexão com um banco de dados, com uma API, um repositório externo ou através de arquivos de texto ou binários. Desta forma, na análise até agora apresentada optou-se por limitar a atenção dedicada a este aspecto de workflows, apresentando apenas os tempos de leitura obtidos por ingerir o conjunto de dados no seu formato original, um arquivo de texto no formato CSV comprimido. No entanto, em aplicações que lidam com grandes volumes de dados, a ingestão de dados tipicamente não seria feita por meio de arquivos de texto, enquanto um dos formatos alternativos citados acima como a leitura de um arquivo binário ou mesmo a partir de um banco de dados seriam candidatos mais adequados. Da mesma forma, caso fosse necessário gerar uma saída com a pipeline, questão até agora desconsiderada nas análises, um arquivo de texto não seria o formato mais adequado.

Desta forma, apenas para trazer visibilidade sobre o impacto que a leitura e a geração de dados por meio de arquivos tem sobre uma pipeline como a analisada neste trabalho, uma análise à parte foi feita sobre os tempos para ler entradas e gerar saídas em cada uma das bibliotecas, usando como referência subconjuntos com $5 * 10^7$ e 10^8 registros. Para trazer visibilidade sobre a influência do formato dos arquivos, também foram realizados experimentos utilizando arquivos parquet, além de csv.

Figura 4.5: Tempo por etapas de I/O com $5 * 10^7$ registros, no formato csv

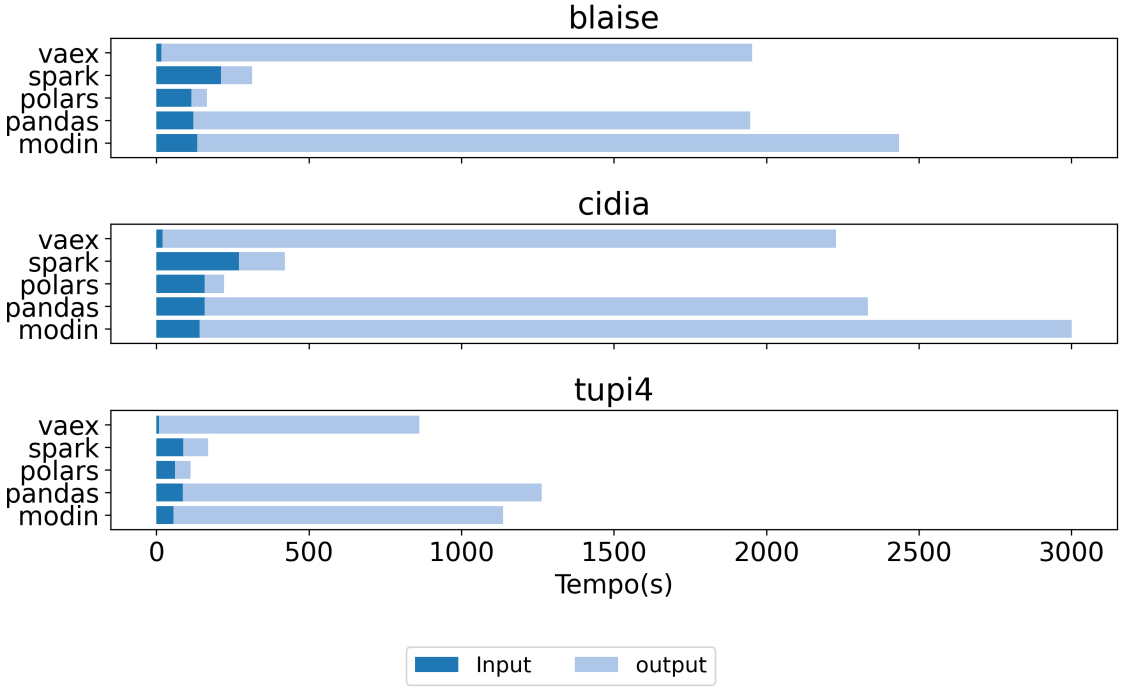
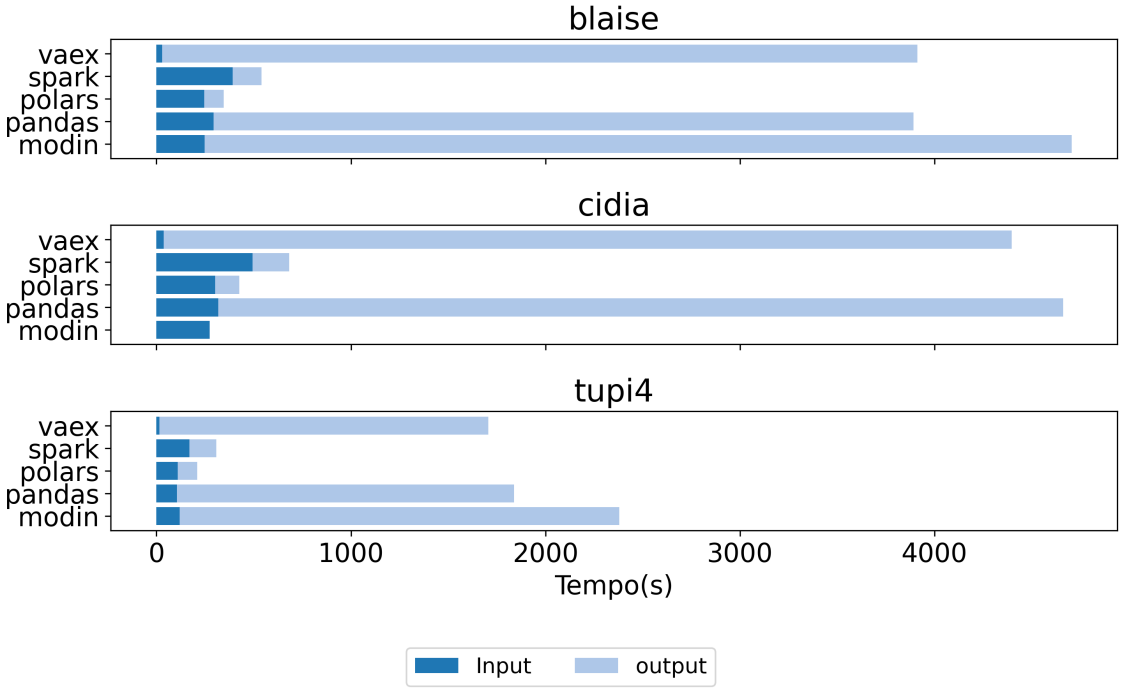


Figura 4.6: Tempo por etapas de I/O com 10^8 registros, no formato csv



As Figuras 4.5 e 4.6 trazem os resultados de I/O utilizando arquivos csv com 5 * 10^7 e 10^8 registros, respectivamente. De imediato, chama a atenção a amplitude dos resultados obtidos: em bibliotecas como o Pandas, Vaex e Modin, o tempo para gerar um arquivo csv excede o tempo de leitura por um fator maior que dez. Enquanto isso, para o Spark e Polars, as duas operações tem durações bem semelhantes, com o Spark sendo a única biblioteca em que o tempo de leitura excede o tempo de geração de arquivos de saída. De maneira geral, pelo tempo que a maioria das bibliotecas leva para ler e gerar extrações em função do tamanho de problema, observa-se uma baixa escalabilidade da utilização deste formato de arquivo.

Por outro lado, as Figuras 4.7 e 4.8 trazem os resultados de I/O utilizando arquivos parquet, também com 5 * 10^7 e 10^8 registros. Nota-se que o uso deste arquivo torna o processo muito mais rápido: o Pandas, onde se obtém o pior resultado na leitura, conclui o processo em média em 63s na Cidia, enquanto o Spark, biblioteca com maior tempo na leitura de csvs, demora 485s em média na mesma máquina. O Vaex novamente se destaca com um excelente desempenho, seguido pelo Spark e Polars, e mantém-se a observação de que gerar arquivos de saída é um processo muito mais lento, desta vez para todas as bibliotecas. Como principal conclusão, considerando-se que os tempos de execução são de outra ordem de grandeza com relação aos da leitura de csvs, o parquet mostra-se uma opção que traz ganhos relevantes em operações de I/O com grandes conjuntos de dados.

Figura 4.7: Tempo por etapas de I/O com 5 * 10^7 registros, no formato parquet

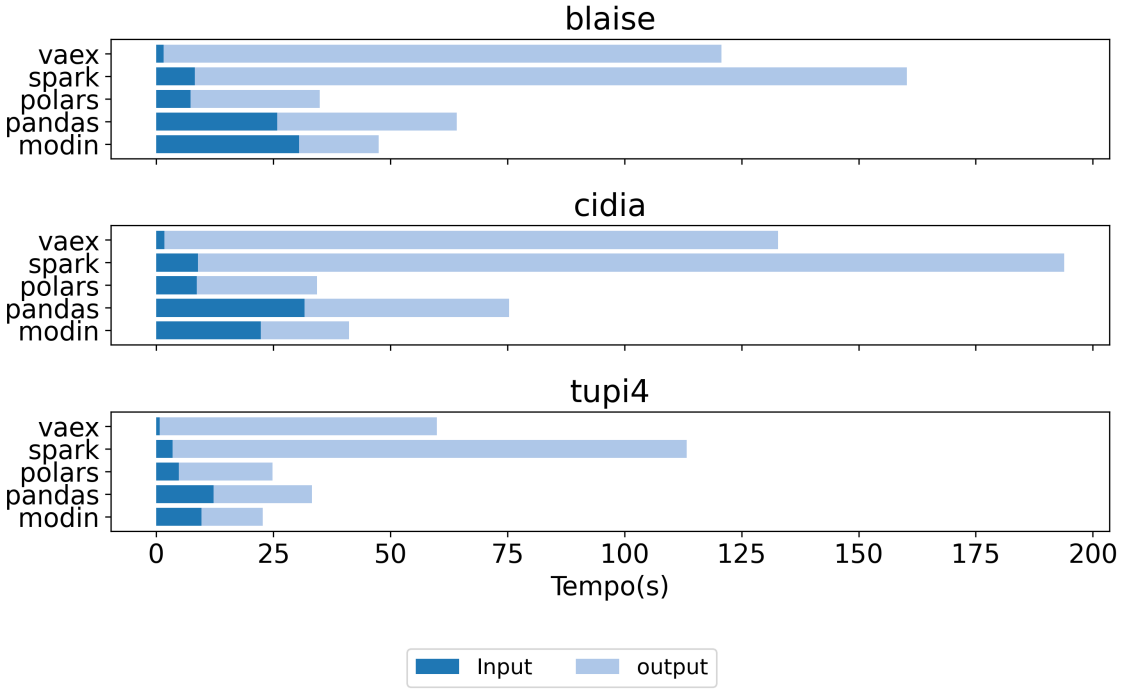
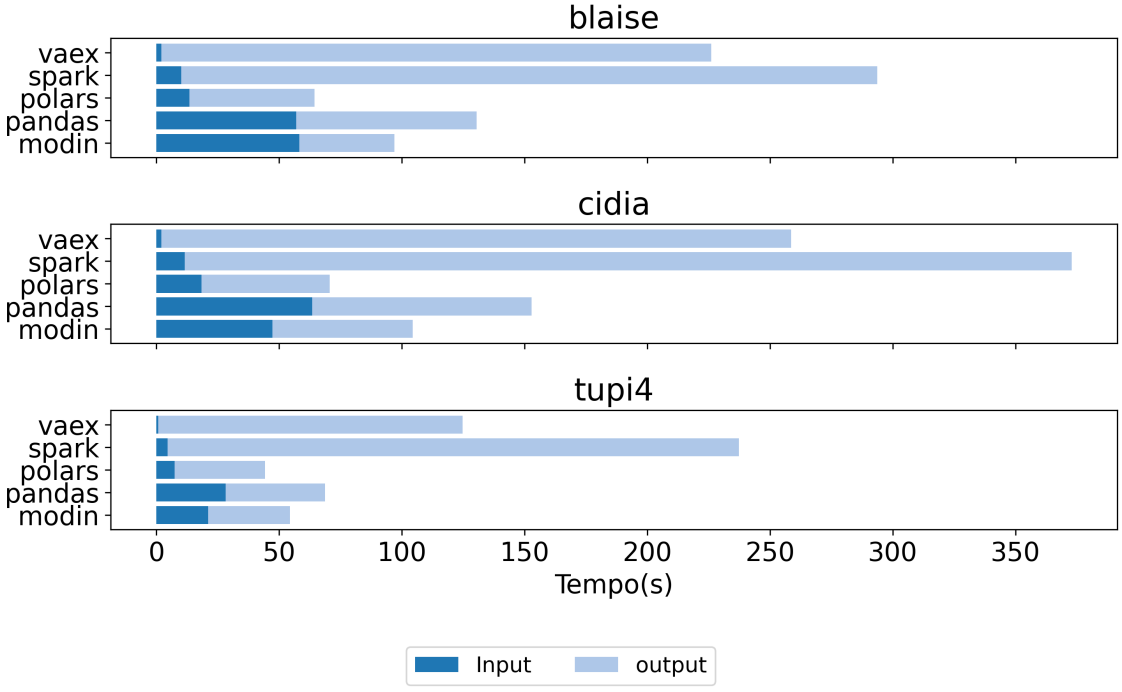


Figura 4.8: Tempo por etapas de I/O com 10^8 registros, no formato parquet

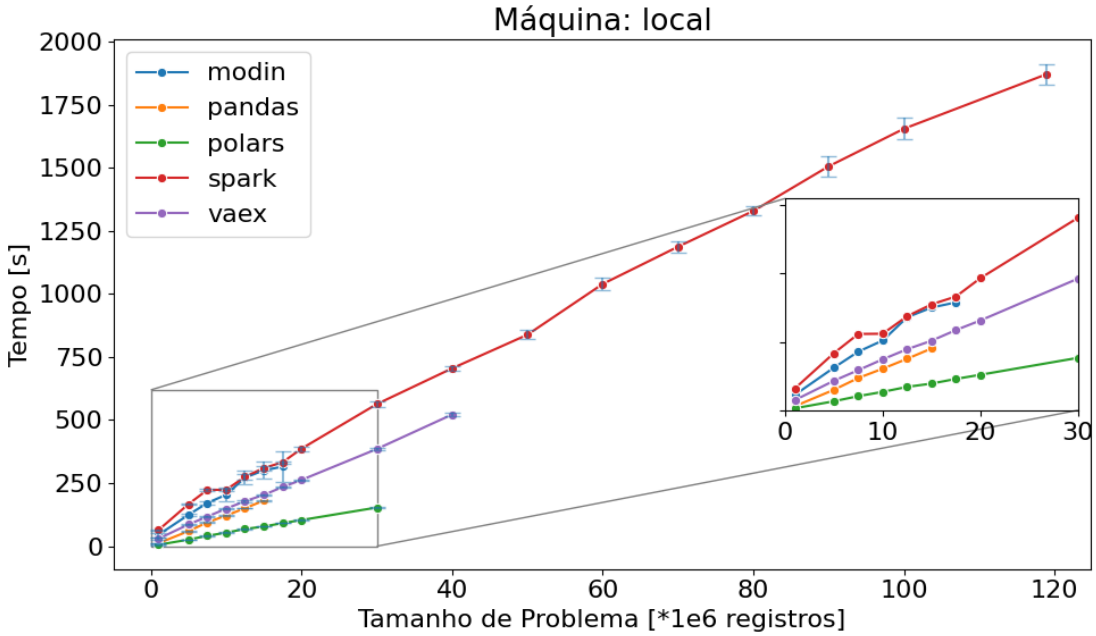


4.3.1 Análise das execuções em Computador Pessoal (PC)

Considerando a grande diferença de recursos existente entre as máquinas do PCAD e o PC utilizado, a análise deste é apresentada à parte, e seus resultados são trazi-

dos na Figura 4.9. Destaca-se o desempenho do Spark, que foi a única biblioteca capaz de processar o conjunto de dados inteiro, resultado bem superior ao das demais: o Vaex teve execuções bem sucedidas com até 4 * 10^7 registros, e o Polars, 3 * 10^7. Este resultado evidencia a importância de considerar o sistema computacional utilizado na aplicação a ser desenvolvida, uma vez que o Vaex e o Polars mostraram-se opções mais rápidas nas máquinas do PCAD, mantendo tal padrão no PC, porém não demonstraram a mesma capacidade de escala que o Spark com recursos mais limitados. Especialmente no caso do Vaex este resultado foi recebido com alguma surpresa, visto que ele é apresentado justamente como uma opção projetada para trazer alta capacidade de escala em computadores pessoais.

Figura 4.9: Tempo de execução em PC, por tamanho de problema



Como a região onde se pode observar mais resultados foi nas medições com subconjuntos menores dos dados, é trazido também na Figura 4.9 um recorte sobre os resultados das execuções de até 3 * 10^7 registros. Em termos das medidas extraídas, nota-se uma menor variabilidade em relação às máquinas de processadores mais rápidos, com a exceção dos últimos pontos do Modin, quando este estava próximo do seu ponto de falha. Para este, seguiu-se observando o mesmo comportamento inesperado de baixo ou nenhum ganho de desempenho ou escalabilidade. Enquanto o Pandas não produziu análises bem sucedidas após a marca de 1,5 * 10^7 registros, o Modin passou a falhar após 1,75 * 10^7 registros, e mesmo com este número apresentou funcionamento irregular, falhando em 40% dos ensaios realizados. No entanto, considerando o que seria esperado do Pandas,

seus resultados podem ser considerados como bastante competitivos, ficando atrás apenas do Polars em tempo total de execução enquanto pode ser utilizado, justificando sua popularidade e aplicabilidade em workflows de menor custo.

Observa-se também que o tempo de inicialização do Spark é muito relevante para os tempos de execução com estas dimensões de conjuntos de dados, em razão da instanciação da JVM e do cluster local que gerenciará a execução, mostrando que o Spark tem seu uso justificado apenas quando existe maior custo computacional, o que, considerando este exemplo de pipeline, se observa a partir dos $3 * 10^7$ registros.

5 CONCLUSÃO

Por meio deste trabalho, explorou-se sob diferentes ângulos o desempenho das bibliotecas de ciência de dados mais populares em python, utilizando o conjunto de dados *Airline on Time* como objeto de estudo. Por meio de uma pipeline hipotética, foram extraídos resultados para as bibliotecas em diferentes sistemas computacionais, destacando as diferentes etapas que tipicamente compõem uma pipeline, e com a sua análise foram destacadas as principais características e vantagens de cada biblioteca, de maneira que os resultados possam servir como insumos para profissionais que se vejam na situação de fazer uma escolha por uma dessas ferramentas. Os principais aprendizados obtidos estão reunidos na Seção 5.1 e, por fim, são apresentadas algumas vias de possíveis trabalhos futuros para aprofundamento do problema a partir deste trabalho.

5.1 Principais aprendizados

Polars e Vaex se destacaram em termos de velocidade de execução, sendo o Vaex o segundo colocado que teria apresentado o melhor resultado, não fosse pelo preparador pivot inexistir na sua API. Dentre estas duas opções, o Vaex se destacou sobretudo na leitura dos arquivos, enquanto o Polars mostrou-se uma opção mais apropriada para aplicar transformações aos dados. Nas demais etapas, ambos se mostraram semelhantes.

Em termos de refatoração de uma base de códigos que utilize Pandas, o Polars novamente aparece como a melhor opção, possuindo API semelhante e documentação abrangente. O Modin naturalmente poderia ser apontado também sob a ótica da facilidade de adoção, mas em termos do seu desempenho e escalabilidade, os resultados aqui obtidos não corroboram sua escolha, salvo novas análises que o suportem.

O SparkSQL mostrou-se a biblioteca mais escalável, sendo a única que executou com sucesso a pipeline no conjunto de dados completo em um ambiente com memória mais reduzida, como o PC. No entanto, mostrou-se também a biblioteca cuja aplicabilidade surge mais tarde do que as outras, à medida que cresce o custo computacional: para problemas menores seu uso tende a não se justificar.

O Pandas mantém-se como uma boa opção para pipelines mais simples, chegando a processar 1,5 * 10^7 registros no PC com bom desempenho, considerando a pipeline utilizada. Para leitura de arquivos, o Vaex destaca-se como a opção de melhor desempenho dentre as analisadas, tanto para parquet quanto CSV.

Caso seja necessário gerar arquivos de saída, tal operação varia muito entre diferentes bibliotecas, portanto é importante fazer a escolha adequada. Para geração de arquivos CSV, o Polars e o Spark mostram-se como opções de melhor desempenho, porém para grandes conjuntos de dados gerar saídas neste formato não é aconselhável. Já para o parquet, formato que ofere melhor escalabilidade, as diferenças entre bibliotecas são menos significativas, com o Pandas, Polars e Modin mostrando-se como as opções com melhor desempenho.

5.2 Possibilidades de trabalhos futuros

Como o presente trabalho explorou o uso de paralelismo apenas em um nó computacional, uma análise mais aprofundada poderia ser feita por meio de execuções em um cluster distribuído de máquinas com as bibliotecas que o suportem, o que viabilizaria também aumentar a escala dos conjuntos de dados analisados. Da mesma forma, o uso de GPUs por meio de frameworks como o rapids (VEGA et al., 2021) traria uma nova abordagem de otimização de desempenho, especialmente relevante dado o enorme interesse que esta tecnlogia tem despertado nos últimos anos.

Como aprofundamento de aspectos específicos aqui abordados, também pode se destacar o problema de I/O, que poderia ser analisado com maior profundidade, trazendo análises sobre as diferentes formas com que se pode carregar e extrair dados de uma pipeline e como cada um influencia o restante da execução como um todo, observando resultados com conjuntos de dados com características diversas. Além disso, novos ensaios com a biblioteca Modin poderiam ser realizados como uma forma de observar a reprodutibilidade e validade dos resultados obtidos, confirmar eventuais limitações da biblioteca ou mesmo problemas nas configurações utilizadas.

REFERÊNCIAS

ARMBRUST, M. et al. Spark sql: Relational data processing in spark. In: **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2015. (SIGMOD '15), p. 1383–1394. ISBN 9781450327589. Available from Internet: <https://doi.org/10.1145/2723372.2742797>.

BEZANSON, J. et al. Julia: A fresh approach to numerical computing. **SIAM review**, SIAM, v. 59, n. 1, p. 65–98, 2017. Available from Internet: <https://doi.org/10.1137/141000671>.

BLANCO, A. F. et al. Asking and answering questions during memory profiling. **IEEE Transactions on Software Engineering**, IEEE, 2024.

BREDDELS, M. A.; VELJANOSKI, J. Vaex: big data exploration in the era of gaia. **Astronomy & Astrophysics**, EDP Sciences, v. 618, p. A13, 2018.

CHAMBERS, J.; HASTIE, T.; PREGIBON, D. Statistical models in s. In: MOMIROVIC, K.; MILDNER, V. (Ed.). **Compstat**. Heidelberg: Physica-Verlag HD, 1990. p. 317–321.

DEVELOPERS, M. **Modin Project**. 2024. Apache License 2.0. Https://github.com/modin-project/modin.

EXPO, D. Airline on time data, 2008. **URL https://doi. org/10.7910/DVN/2**, v. 3, n. 4, p. 5, 2009.

FERNANDES, A. A. et al. Data preparation: A technological perspective and review. **SN Computer Science**, Springer, v. 4, n. 4, p. 425, 2023.

HAMEED, M.; NAUMANN, F. Data preparation: A survey of commercial tools. **SIGMOD Rec.**, Association for Computing Machinery, New York, NY, USA, v. 49, n. 3, p. 18–29, dec 2020. ISSN 0163-5808. Available from Internet: <https://doi.org/10.1145/3444831.3444835>.

HAMEED, M.; NAUMANN, F. Data preparation: A survey of commercial tools. **ACM SIGMOD Record**, ACM New York, NY, USA, v. 49, n. 3, p. 18–29, 2020.

HELLERSTEIN, J. M.; HEER, J.; KANDEL, S. Self-service data preparation: Research to practice. **IEEE Data Eng. Bull.**, v. 41, n. 2, p. 23–34, 2018.

HENRY, L.; WICKHAM, H. **rlang: Functions for Base Types and Core R and 'Tidyverse' Features**. [S.l.], 2024. R package version 1.1.4, https://github.com/r-lib/rlang. Available from Internet: <https://rlang.r-lib.org>.

INFORMÁTICA, I. de. **Parque Computacional de Alto Desempenho**. 2024. Universidade Federal do Rio Grande do Sul. Https://gppd-hpc.inf.ufrgs.br/#orgaa81096.

MCKINNEY, W. et al. pandas: a foundational python library for data analysis and statistics. **Python for high performance and scientific computing**, Seattle, v. 14, n. 9, p. 1–9, 2011.

MODIN, D. **Getting Started with Modin**. 2024. Https://modin.readthedocs.io/en/stable/getting_started/quickstart.html.

MODIN, D. **Modin I/o APIs**. 2024. Https://modin.readthedocs.io/en/stable/supported_apis/io_supported.html.

MORITZ, P. et al. Ray: a distributed framework for emerging ai applications. In: **Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation**. USA: USENIX Association, 2018. (OSDI'18), p. 561–577. ISBN 9781931971478.

MOZZILLO, A.; GAGLIARDELLI, L. **Bento: Dataframes Evaluation Tool**. 2024. Https://github.com/dbmodena/bento.

MOZZILLO, A. et al. Evaluation of dataframe libraries for data preparation on a single machine. **arXiv preprint arXiv:2312.11122**, 2023.

NADI, S.; SAKR, N. Selecting third-party libraries: the data scientist's perspective. **Empirical Software Engineering**, Springer, v. 28, n. 1, p. 15, 2023.

ODERSKY, M. et al. **The Scala language specification**. [S.l.]: Lausanne, Switzerland, 2004.

OURIQUE, J. P. J. **Fork Bento: Dataframes Evaluation Tool**. 2024. Https://github.com/JoaoPedroourique/bento/tree/develop.

PETERSOHN, D. et al. Towards scalable dataframe systems. **CoRR**, abs/2001.00888, 2020. Available from Internet: <http://arxiv.org/abs/2001.00888>.

PETERSOHN, D. et al. **Towards Scalable Dataframe Systems**. 2020. Available from Internet: <https://arxiv.org/abs/2001.00888>.

POLA.RS. **Polars Concepts**. 2024. Https://docs.pola.rs/user-guide/concepts/.

PYPL. **PYPL PopularitY of Programming Language Index**. 2024. Https://pypl.github.io/PYPL.html.

RICHARDSON, N. et al. **arrow: Integration to 'Apache' 'Arrow'**. [S.l.], 2024. R package version 16.1.0, https://arrow.apache.org/docs/r/. Available from Internet: <https://github.com/apache/arrow/>.

ROCKLIN, M. Dask: Parallel computation with blocked algorithms and task scheduling. In: HUFF, K.; BERGSTRA, J. (Ed.). **Proceedings of the 14th Python in Science Conference**. [S.l.: s.n.], 2015. p. 130 – 136.

ROSSUM, G. V.; JR, F. L. D. **Python reference manual**. [S.l.]: Centrum voor Wiskunde en Informatica Amsterdam, 1995.

SCHMITT, M. **Scaling Pandas: Comparing Dask, Ray, Modin, Vaex, and RAPIDS**. 2024. Https://www.datarevenue.com/en-blog/pandas-vs-dask-vs-vaex-vs-modin-vs-rapids-vs-ray.

SHAIKH, E. et al. Apache spark: A big data processing engine. In: **2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)**. [S.l.: s.n.], 2019. p. 1–6.

SINTHONG, P.; CAREY, M. J. Aframe: Extending dataframes for large-scale modern data analysis. In: **2019 IEEE International Conference on Big Data (Big Data)**. [S.l.: s.n.], 2019. p. 359–371.

STANčIN, I.; JOVIć, A. An overview and comparison of free python libraries for data mining and big data analysis. In: **2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)**. [S.l.: s.n.], 2019. p. 977–982.

TEAM pandas development. **pandas dataframe documentation**. 2024. Https://pandas.pydata.org/pandas-docs/stable/ reference/api/pandas.DataFrame.html.

VEGA, J. et al. Reproducible analysis pipeline for data streams: Open-source software to process data collected with mobile devices. **Frontiers in digital health**, Frontiers Media SA, v. 3, p. 769823, 2021.

VINK, R. et al. **pola-rs/polars: Python Polars 1.4.1**. [S.l.], 2024. Available from Internet: <https://doi.org/10.5281/zenodo.13208786>.

WICKHAM, H. et al. **dplyr: A Grammar of Data Manipulation**. [S.l.], 2023. R package version 1.1.4, https://github.com/tidyverse/dplyr. Available from Internet: <https://dplyr.tidyverse.org>.

WU, Y. Is a dataframe just a table? In: SCHLOSS-DAGSTUHL-LEIBNIZ ZENTRUM FÜR INFORMATIK. **10th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2019)**. [S.l.], 2020.

YOO, A. B.; JETTE, M. A.; GRONDONA, M. Slurm: Simple linux utility for resource management. In: FEITELSON, D.; RUDOLPH, L.; SCHWIEGELSHOHN, U. (Ed.). **Job Scheduling Strategies for Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 44–60. ISBN 978-3-540-39727-4.

ZAHARIA, M. et al. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In: **9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)**. San Jose, CA: USENIX Association, 2012. p. 15–28. ISBN 978-931971-92-8. Available from Internet: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>.

ZAHARIA, M. et al. Apache spark: a unified engine for big data processing. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 59, n. 11, p. 56–65, oct 2016. ISSN 0001-0782. Available from Internet: <https://doi.org/10.1145/2934664>.